

Copyright  
by  
Vandana Gambhir  
2015

The Report Committee for Vandana Gambhir  
certifies that this is the approved version of the following report:

**UToobe Media Share Application:  
A Study of Emerging Web Technologies**

APPROVED BY

SUPERVISING COMMITTEE:

---

Adnan Aziz, Supervisor

---

William Bard

**UToobe Media Share Application:  
A Study of Emerging Web Technologies**

by

**Vandana Gambhir, M.B.A.**

**REPORT**

Presented to the Faculty of the Graduate School of  
The University of Texas at Austin  
in Partial Fulfillment  
of the Requirements  
for the Degree of

**MASTER OF SCIENCE IN ENGINEERING**

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2015

Dedicated to my mother-in-law late Mrs. Vidya Devi Gupta, my husband  
Amitabh Gupta and my parents for their unwavering support and patience  
during the course of my M.S.E. Program.

## Acknowledgments

I would like to thank my supervisor, Dr. Adnan Aziz whose guidance and vision made this report possible. He inspired me to explore and learn the new technologies for this project. Special thanks to Prof. William Bard for graciously agreeing to be my reader. Thanks to all my professors at UT, from whom I continually draw support and inspiration.

# **UToobe Media Share Application: A Study of Emerging Web Technologies**

Vandana Gambhir, M.S.E.  
The University of Texas at Austin, 2015

Supervisor: Adnan Aziz

This report studies some of the latest web application development technologies. I have created **UToobe**, a media sharing web application that uses components of the MEAN stack, namely MongoDB, Express and Node.js, as a way of learning these technologies. The application allows users to search for media files uploaded by other users, view them, comment on them, rate them, as well as upload their own media and share it with others.

The primary goal of the project was to identify the challenges faced by Java Developers and explore the benefits and shortcomings of using the new technologies. With a professional background in Java development, I found working with JavaScript and the *callback* oriented style of Node.js to be a challenging paradigm shift. It gave me a completely new perspective on writing web services using REST and JSON. The inbuilt capabilities of frameworks like Bootstrap, and Jade helped give a very professional look and

feel to the application. It was a good educational experience to learn about NoSQL databases, and how they are able to solve the problem of persisting unstructured and large data efficiently.

Based on the experience with the application, I feel the new technologies can cut down development time by two-third as compared to Java technologies. The current application has 1500 lines of code while a Java implementation of the same would have taken around 4500 lines of code. The new technologies also reduce the testing and deployment time by half.

# Table of Contents

<b>Acknowledgments</b>	<b>v</b>
<b>Abstract</b>	<b>vi</b>
<b>List of Tables</b>	<b>xi</b>
<b>List of Figures</b>	<b>xii</b>
<b>List of Sources</b>	<b>xiii</b>
<b>Chapter 1. Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Project Overview . . . . .	2
1.3 Contributions . . . . .	3
1.4 Report Organization . . . . .	3
<b>Chapter 2. Technology Stack</b>	<b>4</b>
2.1 Back-end Technologies . . . . .	4
2.1.1 MongoDB . . . . .	4
2.1.1.1 Review . . . . .	4
2.1.1.2 Structure and Operational Interface . . . . .	6
2.1.1.3 Advantages of MongoDB . . . . .	8
2.1.1.4 GridFS . . . . .	10
2.1.2 Node.js . . . . .	11
2.1.2.1 Event Loop & Non-Blocking I/O . . . . .	12
2.1.2.2 Closure . . . . .	15
2.1.2.3 NPM . . . . .	16
2.1.3 Web Services . . . . .	16
2.1.3.1 REST . . . . .	16



2.1.3.2	JSON . . . . .	17
2.1.4	Express . . . . .	19
2.1.4.1	Express-Session . . . . .	19
2.1.4.2	Body-Parser . . . . .	20
2.1.4.3	Busboy-Body-Parser . . . . .	20
2.1.4.4	MongoDB Node.js Driver . . . . .	21
2.1.4.5	Async . . . . .	21
2.1.4.6	Other Modules . . . . .	22
2.1.5	Multimedia . . . . .	22
2.1.5.1	Fluent-Ffmpeg . . . . .	22
2.1.5.2	GridFS-Stream . . . . .	23
2.2	Front-end Technologies . . . . .	24
2.2.1	Ajax and jQuery . . . . .	24
2.2.2	Jade . . . . .	26
2.2.3	Bootstrap . . . . .	27
2.2.4	Bootstrap-Fileinput . . . . .	28
<b>Chapter 3.</b>	<b>Application Overview</b>	<b>29</b>
3.1	Functionality . . . . .	29
3.2	High Level Architecture . . . . .	30
3.3	Request-Response Pipeline . . . . .	31
3.4	Summary of Technologies Used . . . . .	32
3.5	Data Model . . . . .	33
3.5.1	Media Collection . . . . .	33
3.5.2	User Collection . . . . .	34
3.5.3	fs.chunks Collection . . . . .	34
3.5.4	fs.files Collection . . . . .	35
3.6	Source Structure . . . . .	35
3.7	RESTful Services . . . . .	36

<b>Chapter 4. Implementation</b>	<b>37</b>
4.1 Navigation Bar . . . . .	37
4.2 Authentication . . . . .	38
4.3 Account Creation . . . . .	41
4.4 Dashboard . . . . .	43
4.5 View Media . . . . .	45
4.6 Search . . . . .	47
4.7 Add Media . . . . .	50
4.8 My Media . . . . .	54
4.9 Clip . . . . .	55
4.10 Profile . . . . .	58
4.11 Sign Out . . . . .	59
4.12 Utilities and Configuration Files . . . . .	60
<b>Chapter 5. Conclusion</b>	<b>61</b>
5.1 Lessons & Reflections: An Introspection . . . . .	61
5.1.1 JavaScript . . . . .	61
5.1.2 Node.js . . . . .	62
5.1.3 jQuery-Ajax and Jade . . . . .	63
5.1.4 Bootstrap . . . . .	65
5.1.5 Express . . . . .	65
5.1.6 MongoDB . . . . .	66
5.1.7 Development Environment . . . . .	67
5.1.8 Documentation . . . . .	68
5.1.9 Performance . . . . .	68
5.1.10 Testing . . . . .	68
5.2 Source Code . . . . .	69
5.3 Future Work . . . . .	69
5.4 Summary . . . . .	70
<b>Bibliography</b>	<b>72</b>
<b>Vita</b>	<b>79</b>

## List of Tables

3.1	Summary and Version of Library/Frameworks Used . . . . .	33
3.2	List of RESTful Services consumed by UToobe . . . . .	36
5.1	Media Upload Time(milliseconds) . . . . .	68

## List of Figures

2.1	Event Loop & Non-Blocking I/O . . . . .	13
3.1	High Level Architecture of UToobe . . . . .	30
3.2	Request-Response Pipeline . . . . .	31
4.1	Navigation Bar . . . . .	37
4.2	Login Page . . . . .	39
4.3	Account Creation Page . . . . .	41
4.4	Dashboard Page . . . . .	44
4.5	Comments on Media . . . . .	46
4.6	Search Capability . . . . .	47
4.7	Add Media Page . . . . .	51
4.8	My Media List . . . . .	54
4.9	Media Manipulation . . . . .	55
4.10	Media Clip . . . . .	57
4.11	User Profile . . . . .	58

## Listings

2.1	Storage Structure . . . . .	7
2.2	Collections and Documents . . . . .	7
2.3	Sort Query Operation . . . . .	8
2.4	Storing data in GridFS . . . . .	11
2.5	Blocking I/O Code . . . . .	13
2.6	Non Blocking I/O Code . . . . .	14
2.7	Closure . . . . .	15
2.8	Example of using Express Session . . . . .	20
2.9	Body Parser for JSON . . . . .	20
2.10	Multipart request parser . . . . .	21
2.11	ffprobe used for retrieving media metadata . . . . .	23
2.12	Read and Write Stream using gridfs-stream . . . . .	23
2.13	Example of Ajax POST . . . . .	25
2.14	Example of Jade template and interpolation . . . . .	26
2.15	Example of generated HTML with interpolation . . . . .	27
2.16	Template Engine Configuration with Express . . . . .	27
3.1	Media Document . . . . .	33
3.2	User Document . . . . .	34
3.3	fs.chunk Document . . . . .	34
3.4	fs.file Document . . . . .	35

4.1	Navigation bar with (+) Glyphicon . . . . .	38
4.2	Authentication request . . . . .	40
4.3	Credentials Validation callback . . . . .	40
4.4	Data Store findOne callback . . . . .	40
4.5	Create new user request . . . . .	42
4.6	Create User callback . . . . .	42
4.7	User Data (JSON) and insert callback . . . . .	43
4.8	REST calls for get Recent and Most Likes . . . . .	44
4.9	DB call for get Recent Media . . . . .	45
4.10	Add Comments . . . . .	47
4.11	Search Request . . . . .	48
4.12	Search results (JSON) array . . . . .	48
4.13	Data Store find call . . . . .	48
4.14	Async Waterfall to fetch data from Multiple collections . . . . .	49
4.15	bootstrap-fileinput plug-in . . . . .	50
4.16	jQuery extract form data . . . . .	52
4.17	Create Media Request . . . . .	52
4.18	Multipart request parser with Write request . . . . .	53
4.19	Media data persistence in Data Store . . . . .	53
4.20	Jade Iterator for parsing document array . . . . .	54
4.21	Datastore find, update, remove . . . . .	55
4.22	Vertical flip Request . . . . .	56
4.23	Clip Media Request . . . . .	57

4.24 View Profile callback . . . . .	59
4.25 LogOut callback . . . . .	59
4.26 Database Configuration . . . . .	60

# Chapter 1

## Introduction

### 1.1 Motivation

Web applications are client-server programs used for exchanging information over the Internet. The client-side of these applications typically runs within a browser such as Internet Explorer, Chrome, Mozilla, etc. The client uses the Internet to connect to the server using HTTP [47]. Initial web-applications were made up of mostly static pages. Only a few technologies were available for producing dynamic content. The technology stack mostly consisted of JSP/ASP/JavaScript on the front-end, some J2EE/.NET based server in the middle, ultimately connecting to an RDBMS.

As the cost of data storage dropped, while the speeds of communication increased, web applications began to hit the boundaries of these technologies. Phone based digital cameras became ubiquitous. Social media emerged. And with that came the desire to share a lot of data on the Internet. All the content which was hitherto in the domain of newspaper and television, suddenly shifted to the Internet. Web applications needed to be a lot of more responsive. These applications put too much demands on the web technologies which the current stack was no longer able to fulfill. At the same time, there was emergence of



open-source software. Developers started freely contributing to development of new tools and frameworks. This resulted in an explosion of technologies and software bundles. Some of these tools and frameworks include Motion UI, Web UI components, AngularJS, SPA (Single Page Apps [1]), etc. This is very different from purely Java based programming stack that I have worked with in the past. This report studies these new technologies, namely MongoDB, Express, and Node.js in the context of the web-application **UToobe** development and compares the development effort with Java technologies.

## 1.2 Project Overview

When it comes to web technologies, most of the advancement in recent times has happened in two areas, namely, responsive UI and storage of large-sized unstructured data. **UToobe** concentrates on these two areas in the context of an online media sharing application. **UToobe** has been implemented on full stack JavaScript using Node.js, MongoDB, Express, Jade, jQuery and REST APIs. The application allows users to create their profile, upload videos, images and other media, search and watch other user's media, manipulate uploaded media, rate their postings and add comments. Bootstrap and Jade made up the responsive UI. It helped me to learn the event-driven non-blocking I/O paradigm of Node.js. Data persistence has been done using MongoDB, one of the most popular NoSQL databases.

### 1.3 Contributions

- Designed, implemented and tested **UToobe**, a media sharing web application.
- Reported on benefits and pitfalls of new web technologies like Node.js, Express, Jade, Bootstrap and MongoDB from the perspective of a Java developer.
- Made the source code and documentation for the application available freely as open source.

### 1.4 Report Organization

Multiple front-end and back-end technologies like MongoDB, Node.js, Express were used to develop **UToobe**. Chapter 2 provides a brief introduction to these technologies, their strengths and weaknesses. Chapter 3 provides an overview of the application in terms of its functionality. This chapter also explains the system architecture, request pipeline, data model and how the various technologies were used. Chapter 4 discusses the implementation details. It describes the main use cases and how they were realized, together with code snippets. Chapter 5 summarizes the report, talks about future work and closes with conclusions.

# Chapter 2

## Technology Stack

This chapter describes the software technologies used in order to achieve the goals of the project. These technologies are part of the MEAN stack. I specifically used MongoDB, Express and Node.js.

### 2.1 Back-end Technologies

#### 2.1.1 MongoDB

##### 2.1.1.1 Review

With the increase in communication speeds and the drop in communication and storage costs, the size of data that can now be stored in computer applications has exploded. Growth of social media and ubiquity of phone-cameras has also made it easy for users to record and share their media files (audio/video, pictures, etc.) online which require large storage. Although relational databases have been very successful in storage and retrieval of data for enterprise applications, these Internet applications posed a new challenge. Size of data was huge and it was not as well organized as business data, and could not be properly stored in the tabular/relational structure of relational databases. RDBMS [58](Relational Database Management System) offered the facility to store large files in the form of LOBS [11](Large Objects) but

their storage and retrieval was both cumbersome as well as slow. RDBMS also do not provide an efficient way to distribute data as well as the storage/retrieval tasks across multiple servers.

This requirement led to the popularity of a different type of databases, called NoSQL [9] databases. MongoDB [33] used in my application is one of the most popular NoSQL databases. NoSQL (also known as Non-SQL or Non-relational) databases were first conceived in 1960s but did not become popular until Web 2.0 came along some 10 years back. Unlike RDBMS, NoSQL databases do not store information in a tabular format and instead use a key-value, graph or a document based data-structure. The particular suitability of a given NoSQL database depends on the problem it must solve. Choice of the database would depend on the type of data and the application. For example, for an enterprise application that deals with structured data, transactional operations and multiple updates, RDBMS might be a better choice than NoSQL. On the other hand, for social media applications that have unstructured data, have many optional fields, have very few updates and deal with storing large media files, a NoSQL database will be more appropriate.

Typically NoSQL databases compromise consistency in order to provide better availability, partition tolerance, and speed. NoSQL databases also provide simplicity of design, simpler *horizontal* scaling to clusters of machines, which is a problem for relational databases, and finer control over availability. The data structures used by NoSQL databases are also viewed as *more flexible* than relational database tables. For example, in typical NoSQL database like

MongoDB, a new attribute can be added to the store on the fly, without first altering the table structure [15]. NoSQL databases are increasingly used in big data [42] and real-time web applications. NoSQL systems are also sometimes called *Not only SQL* to emphasize that they may support SQL-like query languages [61].

NoSQL databases do have some drawbacks. They require use of low-level query languages (instead of SQL to perform ad-hoc JOINS across tables) and lack standardized interfaces. Most NoSQL stores lack true ACID [43] transactions. Instead they offer a concept of *eventual consistency* in which database changes are propagated to all nodes *eventually* (typically within milliseconds) so queries for data might not return updated data immediately.

MongoDB [56] is one of the popular NoSQL databases that I chose to use in the project. It is a cross-platform document-oriented database. MongoDB uses JSON-like documents [5] with dynamic schemas (called BSON [2]), making the integration of data in certain types of applications easier and faster [37].

#### 2.1.1.2 Structure and Operational Interface

**Storage Structure:** MongoDB [33] stores data in the form of documents, which are JSON-like field and value pairs [5]. Documents are analogous to structures in programming languages that associate keys with values (e.g., dictionaries, hashes, maps, and associative arrays). Formally, MongoDB documents are BSON documents. BSON [2] is a binary representation of JSON

with additional type information. In the documents, the value of a field can be any of the BSON data types, including other documents, arrays, and arrays of documents. For more information, refer to Listing 2.1:

```
{
  name : "tom",           // field : value
  age  : 10,              // field : value
  grade : "A",
  interest : [ "news", "sports" ]
}
```

Listing 2.1: Storage Structure

MongoDB stores all documents in collections. A collection is a group of related documents that have a set of shared common indexes [44]. Collections are analogous to a table in relational databases and a row to a document in the collection. For more information, refer to Listing 2.2.

```
UsersCollection
[
  doc#1 {
    name : "tom",           // field : value
    age  : 10,              // field : value
  }
  doc#2 {
    name : "jane",
    age  : 11,
  }
  doc#3 {
    name : "joe",
    age  : 11,
  }
]
```

Listing 2.2: Collections and Documents

## Database Operations:

- **Query Operations:** In MongoDB a query targets a specific collection of documents. Queries specify criteria, or conditions, that identify the documents that MongoDB returns to the clients. A query may include a projection that specifies the fields from the matching documents to return. You can optionally modify queries to impose limits, skips, and sort orders. In the following Listing 2.3, the query process specifies a query criteria and a sort modifier.

```
// Query Criteria : age > 12.  
// Sort by age ascending.  
db.userCollection.find( { age: {$gt:10}}).sort({  
    age:1})
```

Listing 2.3: Sort Query Operation

- **Data Modification Operations:** Data modification refers to operations that create, update, or delete data. These operations modify the data of a single collection. For the update and delete operations, one can specify the criteria to select the documents to update or remove.

### 2.1.1.3 Advantages of MongoDB

Following are some of the main advantages [56] of MongoDB and reasons why I chose MongoDB for this application:

- It has a number of resources and drivers (client libraries) available to handle the interaction with database in a language appropriate to the application.
- It has a small learning curve and suitable for project timelines.

- It is available for free under the GNU Affero General Public License and installation is simple.
- It uses JSON-like documents with dynamic schemas. Since my application heavily uses JavaScript, a JSON based interface would be a natural choice.
- It is designed for OLTP workloads and provides the ability to do complex queries.
- It enables horizontal scalability by using a technique called *sharding*. Sharding distributes the data across physical partitions to overcome the hardware limitations. The data is automatically balanced in the clusters.
- It also provides ACID [43] properties at the document level as in the case of relational databases.
- It supports replica sets, in other words, a failover mechanism is automatically handled. If the primary server goes down, the secondary server becomes the primary automatically without any human intervention.
- It supports the common authentication mechanisms, such as LDAP [51], AD [38], and certificates. Users can connect to MongoDB over SSL [59] and the data can be encrypted.
- It can be a cost effective solution as it is flexible and reduces cost of hardware and storage.

However, MongoDB has some disadvantages. It is not suitable for complex transactions or for applications that have reporting-style workloads. Unlike HBase [41], it does not provide easy support for Hadoop [45].



#### 2.1.1.4 GridFS

Efficient storage and retrieval of large media files is critical for this application, as is for any social media application. MongoDB is helpful in that via GridFS [33]. GridFS is a specification for storing and retrieving files that exceed the BSON [2] document size limit of 16MB. Instead of storing a file in a single document, it divides a file into parts, or chunks, and stores each of those chunks as a separate document. By default it limits chunk size to 255 kilobytes.

**GridFS Collections:** It uses two collections to store files. One collection stores the file chunks, and the other stores file metadata. When one queries a GridFS store for a file, the driver or client will reassemble the chunks as needed. A range of queries can be executed on files stored. One also can access information from arbitrary sections of files, for example, *skipping* into the middle of a video or audio file. To store and retrieve files using GridFS, I used a MongoDB Node.js driver and gridfs-stream module.

Collections are placed in a common bucket by prefixing each with the bucket name. By default, it uses two collections with names prefixed by *fs* bucket, namely, *fs.files* and *fs.chunks*. One can also choose a different bucket name than *fs*. Each file chunk is identified by its unique ObjectId stored in its *\_id* field. GridFS uses a unique, compound index [44] on the chunks collection for the files *\_id* and *and* fields.

The application stores media data in GridFS. The following code-snippet shows how large files are stored/retrieved using GridFS using `gridfs-stream` [29]

```
//gfs : instance of gridfs-stream
var writeStream = gfs.createWriteStream({
  _id: fileId,
  filename: filename,
  mode: 'w',
  content_type: mimetype,
});
file.pipe(writeStream);
}).on('finish', function() { ..... });
```

Listing 2.4: Storing data in GridFS

### 2.1.2 Node.js

Node.js [7] is an open-source software hosted and supported by the Node.js Foundation and is a collaborative project at Linux Foundation. Simply put, it provides a JavaScript runtime environment. It runs on Google's V8 Engine. With the help of inbuilt and external libraries, Node.js provides much more. For instance, Node.js contains a built-in library to allow applications to act as a stand-alone web server. It provides the ability to create real-time, two-way connection applications where both the client and server can initiate communication and exchange data. A large percentage of the basic modules are written in JavaScript. It is a cross-platform runtime environment for developing server-side web applications.

### 2.1.2.1 Event Loop & Non-Blocking I/O

What sets Node.js apart is its event-driven architecture and a non-blocking I/O design to optimize an application's throughput and scalability, especially for real-time web applications. In most languages like C or Java, a system's performance can be enhanced by using a higher number of threads. When one thread is blocked for I/O, other threads can make use of the CPU time and provide better throughput. However, that results in wasted CPU cycles due to thread-context switching as well allocation of memory and other resources to each thread.

Ultimately, only one thread can execute within a core at a time [18]. Multithreading helps performance by scheduling another thread for processing while one thread is waiting for I/O. Node instead uses a single main thread to service all requests while requiring that I/O activities be non-blocking, thereby making sure I/O calls are not preventing optimum CPU utilization. This is a much simpler approach. It requires that any function performing I/O must use a callback. This means that HTTP requests, database queries, file I/O, and other things that require a program to wait for a system call, do not force the main thread to wait for the call completion. Instead, they run independently, and then emit an event when the call is completed.

JavaScript runtimes contain a message queue (Figure 2.1) which stores a list of events to be processed and their associated callback functions. In a loop, the queue is polled for the next message (each poll referred to as a *tick*) and when a message is encountered, the callback for that message is executed.

Sharing a single thread between all the requests enables it to support highly concurrent applications. In order to accommodate the single-threaded event loop, it utilizes the libuv library which in turn uses a fixed-sized thread-pool that is responsible for all non-blocking asynchronous I/O operations. As soon as event-loop encounters a system call, the I/O operation is delegated to the thread-pool while the main thread goes on to process next task from the queue.

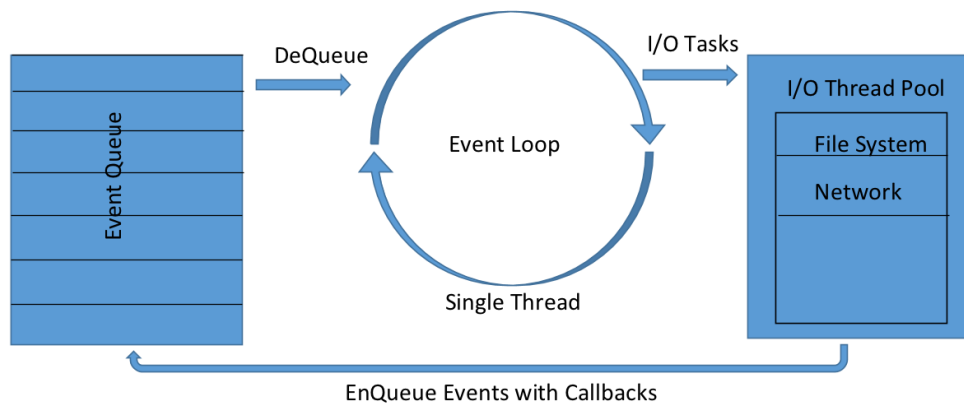


Figure 2.1: Event Loop & Non-Blocking I/O

Let's first see how a regular blocking I/O code works. In the following code, we are retrieving a team's ranking from database and then displaying it on console. :

```
function getTeamRank(teamname) {  
    var team = db.query(teamname);  
    return team;  
}  
console.log('Team Rockets Rank is' + getTeamRank('Rockets').rank);
```

Listing 2.5: Blocking I/O Code

In this case, the function `getTeamRank()` blocks until the database query is complete. As a result, the thread executing this code will block until the function returns. In Node.js, same functionality will be implemented differently. Instead of waiting for the query to finish, the query code is passed a callback function to be invoked when the query is complete. This leaves the executing thread free to execute another task from the queue while the query executes in a separate thread. To realize this, the code makes use of an async function. (Listing 2.6)

```
function getTeamRank (teamname, callback) {  
    db.query(teamname, callback);  
}  
function display(team) {  
    console.log(team.rank);  
}  
getTeamRank ("Rockets", display);
```

Listing 2.6: Non Blocking I/O Code

In Listing 2.6, `getTeamRank()` is an async function. All async functions accept a callback as a last parameter and the callback function accepts error as a first parameter. Call to function `getTeamRank()` returns immediately without waiting for database query(`db.query()`) to finish, freeing the execution thread for other tasks. The callback function `display()` is invoked once the query finishes.

A downside of this single-threaded approach is that Node.js does not utilize multiple cores of the machine unless we use additional modules such as `cluster`, `StrongLoop Process Manager`, or `pm2`. Developers can increase the default number of threads in the `libuv` threadpool. These threads will be

delegated to available cores by the operating system. Another downside is that programming in Node.js has lots of callbacks to deal with non-blocking I/Os. Callbacks often initiate other callbacks in a cascading fashion. This makes code difficult to understand and debug.

#### 2.1.2.2 Closure

Closure is a powerful programming concept in Node.js that helps in data-encapsulation and loose coupling. Simply put, a closure is a function that is nested within the scope of another function. The inner function has access to the outer function's variables as well as the outer function's parameters. As such, the inner function uses the values of variables of the outer function as they are happen to be at the time of execution. The state of the external function is stored as a variable. Below, is a small example that explains this concept:

```
function currentYear (currYear) {
  var nameIntro = "This celebrity is ";
  // this inner function has access to the outer
    function's variables, including the parameter
  function calcAge (yearofBirth) {
    return "Age is " (currYear - yearofBirth);
  }
  return calcAge;
}
var thisYear = currentYear (2015);
thisYear.calcAge(1980); // Returns 'Age is 35' as it
    uses the value 2015 for currYear.
```

Listing 2.7: Closure

### **2.1.2.3 NPM**

NPM [57] stands for Node Package Manager. It is bundled and automatically installed with Node.js. It is written entirely in JavaScript. It provides a command line interface and helps manage dependencies for an application using a manifest file called *package.json*. It also allows users to connect to NPM registry to download, upload, publish and share libraries. This enables easy access to the most commonly used and powerful modules like express, async, mocha, underscore, stringify, etc.

### **2.1.3 Web Services**

#### **2.1.3.1 REST**

Web services are client and server applications that communicate over the World Wide Web using HyperText Transfer Protocol (HTTP). REST (Representational State Transfer) and SOAP (Simple Object Access Protocol) are the two main software architectural styles for implementation of web-services. Since Web Services interact using text based formats (mostly XML, JSON or plain-text) they are language and platform independent. SOAP is an XML-based protocol and consists of envelope and a body.

Typically, the interface of a SOAP based web service is defined in term of a WSDL (WebService Description Language), an XML based interface definition language. A WSDL defines the methods that can be invoked by the clients as well as the structure of input/output parameters in the form of XML schemas (XSDs). SOAP is very bulky and verbose since it is XML based. Con-

suming SOAP services can become extremely complex and cumbersome if the programming language does not have inbuilt support for it.

In comparison, REST provides a lightweight alternative. In REST Architecture everything is a resource. Instead of using XML to invoke remote methods, a REST client relies on URIs (Universal Resource Identifiers) to query the state of a particular resource. In response, the RESTful web service returns a representation of the requested resource. Any type of text format can be used for data-exchange, such as plain text, JSON, XML, etc.

It is important to note that REST is not a standard but more of a guideline. RESTful web services are easier to learn, develop and consume. For these reasons, I picked it for the design of the application.

REST also has some disadvantages in that it does not cover all web services standards, like Transactions, Security, Addressing, Trust, Coordination, etc. REST requests (especially GET method) are not suitable for large amount of data. There is no direct support for generating a client from server-side-generated metadata.

#### **2.1.3.2 JSON**

The first wave of web applications and distributed systems was accompanied by the popularity of XML as a format for data exchange between the applications. It is a human readable format, is technology/platform independent, can be associated with schema definitions thereby enabling validation, and supports hierarchical structures. However, it has some drawbacks, e.g.,



it can become very bulky and does not render itself to easy consumption by software programs. These points together with the popularity of JavaScript for writing web application gave rise to JSON (JavaScript Object Notation). JSON is a lightweight data-interchange format. Like XML, it is a human readable format, but at the same time it is also easy for machines to parse and generate it. Unlike XML, it is very compact and concise. It supports hierarchical structures. Although it is language independent, JSON is a natural choice for JavaScript based applications since it can be automatically parsed/generated by JavaScript without the need for any developer written code. Libraries exist to do the same for other languages like Java, C, etc. JSON is built on two structures:

- A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

These are platform/language independent universal data structures. It makes sense that a data format that is interchangeable with programming languages also be based on these structures. For these reasons I chose to use JSON for my application.

### 2.1.4 Express

Express [3] is an application server framework that is used for building web and mobile applications. It is the de facto standard server framework for Node.js. It is the back-end part of the MEAN stack, together with MongoDB database. It facilitates a rapid development of Node.js based web applications. Following are some of the core features of Express framework:

- Acts as an HTTP server and respond to HTTP Requests (GET, HEAD, PUT and DELETE).
- Defines a routing table which is used to perform different action based on HTTP verb and URL.
- Allows dynamically rendering of HTML pages via configured middleware.
- Cookies management is available via cookie-parser middleware.
- Provides an easy integration with Database with use of appropriate Node.js driver.

#### 2.1.4.1 Express-Session

In any web application, we need to track state of a user's activity as the user traverses from one page to another. This is done through session management. Express Sessions [26] is a Node.js module that helps web applications to maintain user sessions. Session data is stored on the server-side as session ID. MemoryStore is the default server-side session storage. Following code snippet creates and associates a session object with request.

```
app.use(session(
  {secret: 'mySecret', cookie: {maxAge:
    3600000},
  resave: false, saveUninitialized: false}
))
```

Listing 2.8: Example of using Express Session

#### 2.1.4.2 Body-Parser

Body-Parser [23] provides body parsing mechanism to request objects. The module provides the following capabilities: 1) JSON body parsing 2) Raw body parsing 3) Text body parsing 4) URL-encoded form body parsing. Middleware populates *req.body* property with the parsed body or provides an error to the callback. This, however, does not handle multipart bodies due to their complex and typically large nature.

```
// parse application/json
app.use(bodyParser.json());
```

Listing 2.9: Body Parser for JSON

#### 2.1.4.3 Busboy-Body-Parser

Busboy-Body-Parser [25] is a module that provides ability to parse the body of multipart form-data. The parser adds regular fields to `HttpRequest` object (*req.body*) as per body-parser but also adds uploaded files to *req.files*. The limit of the file size can be defined which is applied to individual files rather than the total body size. Example of format of *req.files* is given below.

```

req.files:
{
  fieldName: {
    data: "raw file data",
    name: "upload.txt",
    encoding: "utf8",
    mimetype: "text/plain",
    truncated: false
  }
}

```

Listing 2.10: Multipart request parser

#### 2.1.4.4 MongoDB Node.js Driver

The MongoDB [30] Node.js driver is the officially supported Node.js driver for MongoDB. It is written in pure JavaScript and provides a native asynchronous Node.js interface to MongoDB [21]. The driver is optimized for simplicity. Some of the calls that it provides for CRUD operations are `collection.insert()`, `collection.find()`, `collection.findAndModify()`, `collection.update()` and `collection.delete()`.

#### 2.1.4.5 Async

Async library is designed to control execution flow of an asynchronous program. There are a number of such libraries available, namely, Async, Step, Node Fibers and Syncify. I used Async.js [22] which is the most popular async library in the JavaScript Community. It is available for Node.js as well as for browsers. It has a simple design and implements several methods that one can combine to achieve a parallelism or a composition model. It provides the

ability to control asynchronicity and facilitates serial, parallel, waterfall and queued execution. I used two functions, namely, `.each` and `.waterfall`.

- **.each** : This is a loop for executing asynchronous task. Each task can be called in parallel using this function.
- **.waterfall** : Modules written in waterfall model pass data to next module in a waterfall pattern.

#### 2.1.4.6 Other Modules

Morgan [31] is an HTTP request logger middleware for Node.js. This logger middleware provides the ability to configure the format and options of the output. Available format configurations includes - combine, tiny, common, dev and short. By default the output is STDOUT but it can be configured to output to file system. Nodemailer [32] module is used to send emails. It supports different transport methods, namely, SMTP, sendmail, etc. I used the preconfigured services for using SMTP with Gmail.

### 2.1.5 Multimedia

#### 2.1.5.1 Fluent-Ffmpeg

FFmpeg [4] is a free software project that produces libraries for handling multimedia data. It includes libavcodec (an audio/video codec library) and libavformat (a generic framework for multiplexing and demultiplexing). It provides command line program for transcoding multimedia files. FFmpeg is published under GNU General Public License 2+. The name of the project is inspired by *FF* for *fast forward* and the MPEG video. FFmpeg is comprised

of various components - ffmpeg, ffserver, ffplay, ffprobe, etc.

I used fluent-ffmpeg [28] module. This provides an abstraction of complex command-line usage of ffmpeg and ffprobe into a fluent, easy to use Node.js module. This library was used to manipulate media, namely, horizontal and vertical flips, reading media metadata and clipping it.

```
ffmpeg()
  .input('/path/to/image.jpeg')
  .ffprobe(function(err, data) {
    console.log('image metadata:');
    console.dir(data);
  });
```

Listing 2.11: ffprobe used for retrieving media metadata

#### 2.1.5.2 GridFS-Stream

Gridfs-Stream [29] is a Node.js module that provides streams for writing to and reading from GridFS on MongoDB. Following code creates and reads stream.

```
// streaming to gridfs
var writestream = gfs.createWriteStream({
  filename: 'my_file.txt'
});
fs.createReadStream('/some/path').pipe(writestream);
// streaming from gridfs
var readstream = gfs.createReadStream({
  filename: 'my_file.txt'
});
```

Listing 2.12: Read and Write Stream using gridfs-stream

## 2.2 Front-end Technologies

Today, there are multiple front-end frameworks that make web development faster and easier. The frameworks are modular and lightweight making it easy for developers to mix components. Each framework however, has its own strengths/weaknesses and specific areas of application. This allows one to choose the tools based on the needs of a project.

### 2.2.1 Ajax and jQuery

Ajax stands for asynchronous JavaScript and XML. It is a set of web development techniques used on the client-side to create asynchronous Web applications. With Ajax, web applications can send and receive data from a server asynchronously without interfering with the display and behavior of the existing page. Data can be sent or retrieved using the XMLHttpRequest object. Despite the name, the use of XML is not required (JSON is often used in the AJAX variant), and the requests do not need to be asynchronous. HTML and CSS can be used in combination to mark up and style the information. JavaScript and jQuery are used for creating Ajax request and the response is used for dynamically update web pages. It provides a way for exchanging data asynchronously between browser and server without full page reloads. This greatly enhances the user experience. The Google search type ahead feature is an everyday example of Ajax in action.

jQuery is a cross-platform JavaScript library designed to simplify the client-side scripting of HTML. jQuery, at its core, is a DOM (Document Object

Model) manipulation library. The DOM is a tree-structure representation of a web page. jQuery simplifies the syntax for finding, selecting, and manipulating DOM elements. For example, jQuery can be used for finding an element in the document with a certain property (e.g., all elements with an h1 tag), changing one or more of its attributes (e.g., color, visibility), or making it respond to an event (e.g., a mouse click). It can also be used to create animations as well as Ajax applications. It is an open-source software licensed under the MIT License. It also provides capabilities for developers to create plugins on top of the JavaScript library [12]. It is possible to perform browser-independent Ajax queries using *\$.ajax* and associated methods to load and manipulate remote data.

```
$.ajax({
    type: "POST",
    url: "/user/city",
    data: "name=John&location=Austin"
})
.done( function(msg) { alert( "Data Saved: " + msg
    ); })
.fail( function( xmlHttpRequest, statusText,
    errorThrown )
{
    alert("Your form submission failed.");
}
);
```

Listing 2.13: Example of Ajax POST

The example (Listing 2.13) posts the data name=John and location=Austin to URI `/user/city` on the server. When this request finishes, the success function is called to alert the user. If the request fails, it will alert the user to



the failure. Note that the above example uses the deferred nature of \$.ajax() to handle its asynchronous nature: .done() and .fail() create callbacks that run only when the asynchronous process is complete.

### 2.2.2 Jade

A template engine is a library or framework that uses some language/rules to interpret the data coming from the back-end and rendering HTML views. There are a number of JavaScript template engines available today, namely, Jade, Mustache.js, Dust.js, EJS, etc., Jade being the most popular of them. It is heavily influenced by Haml [46] and is implemented with JavaScript for Node.js and browsers. It has a clean, whitespace-sensitive syntax for writing HTML with no requirement for closing tags. It has an extensive layout inheritance. It supports use of locals or variables, setting of attribute values, scripts, style blocks, conditions, iterations, filters, interpolations and includes. Variables inside templates are either passed from the server-side code or set into the template [17].

```
1 | input { "name": "John" }
2 | /***** JADE template: *****/
3 | doctype html
4 | html
5 |   head
6 |     title Example of a Jade Template
7 |   body
8 |     h1 Hello #{name}
```

Listing 2.14: Example of Jade template and interpolation

Listing 2.14 and 2.15 displays Jade interpolation example.

```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Example of a Jade Template</title>
5   </head>
6   <body>
7     <h1>Hello John</h1>
8   </body>
9 </html>

```

Listing 2.15: Example of generated HTML with interpolation

A view engine can be set up in Express using the following code.

```
express.set("view engine", "jade");
```

Listing 2.16: Template Engine Configuration with Express

### 2.2.3 Bootstrap

Bootstrap is a framework designed for building responsive, accessible and mobile-ready websites. It is an open-source platform. It uses HTML5, CSS3, and jQuery. It contains HTML and CSS-based design templates for typography, forms, buttons, navigation and other interface components, as well as optional JavaScript extensions. It features a 12-column grid and icons that one can incorporate into the user interface.

Bootstrap is compatible with the latest versions of Google Chrome, Firefox, Internet Explorer, Opera, and Safari browsers. Since version 2.0 it also supports responsive web design. This means the layout of web pages adjusts dynamically, taking into account the characteristics of the device used (desktop, tablet, mobile phone). Starting with version 3.0, Bootstrap adopted

a mobile first design philosophy, emphasizing responsive design by default. I used Bootstrap's CSS and components to build the front-end of the application.

#### **2.2.4 Bootstrap-Fileinput**

Bootstrap-Fileinput is an enhanced HTML 5 file input [24] plugin with the ability to preview files and offers multiple files selection. The plugin allows one to setup an advanced file picker/upload control. It is built to work with Bootstrap CSS3 styles. It enhances the file input functionality by offering support to preview a wide variety of files, i.e., images, text, html, video, audio, flash, and objects. In addition, it provides Ajax based uploads, dragging and dropping files, viewing upload progress, and selectively previewing, adding, or deleting files. I used this plugin in my application give user the ability to select/drag-drop media files for adding to the data store.

## Chapter 3

# Application Overview

### 3.1 Functionality

UToobe is a web-based application that allows users to view and share media online. It provides the following functionality:

- Create unique users by choosing login ID and password.
- View and edit user profile.
- Create social-circles and restrict media share.
- Upload media files together with their title and keywords.
- Search for media files that match based on keywords, title, author and category.
- View/play media files.
- Add comments to media submitted by other users.
- Rate media submitted by other users.
- Manipulate media.
- View most recent/highly voted media.

## 3.2 High Level Architecture

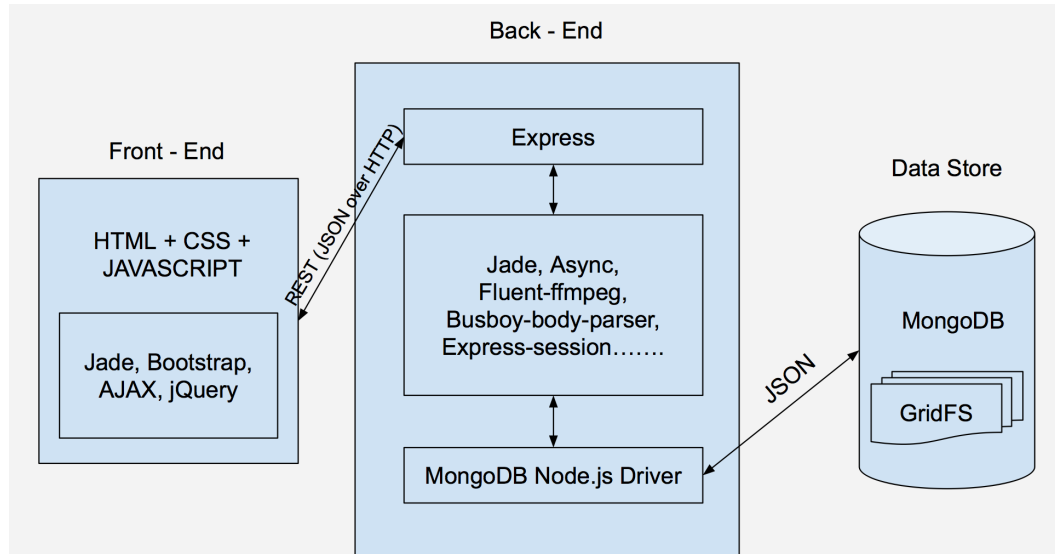


Figure 3.1: High Level Architecture of UToobe

The application makes use of three of the components of the MEAN stack, namely, MongoDB [34], Express [3] and Node.js [7]. Instead of AngularJS [40], my application uses jQuery [50] and Jade [14]. MongoDB provides an easy, JSON oriented interface for storing large, unstructured data in a NoSQL persistent store. Express is the lightweight HTTP Server module that receives the web-requests and performs routing, redirection and caching. It also supports configuration of template engines for the *view*. Node.js provides a server platform for non-blocking I/O and callbacks for executing JavaScript in headless mode. It makes available a library of modules like busboy-body-parser, fluent-ffmpeg, express-session, morgan, gridfs-stream and MongoDB driver that were used for implement the functionalities. jQuery [50] library is

used for making Ajax [39] requests, manipulating elements, triggering and listening for events, selecting elements from the DOM, getting and setting form input values and traversing the DOM. This worked well with template engine Jade and eased the task of interacting with elements on a web page.

### 3.3 Request-Response Pipeline

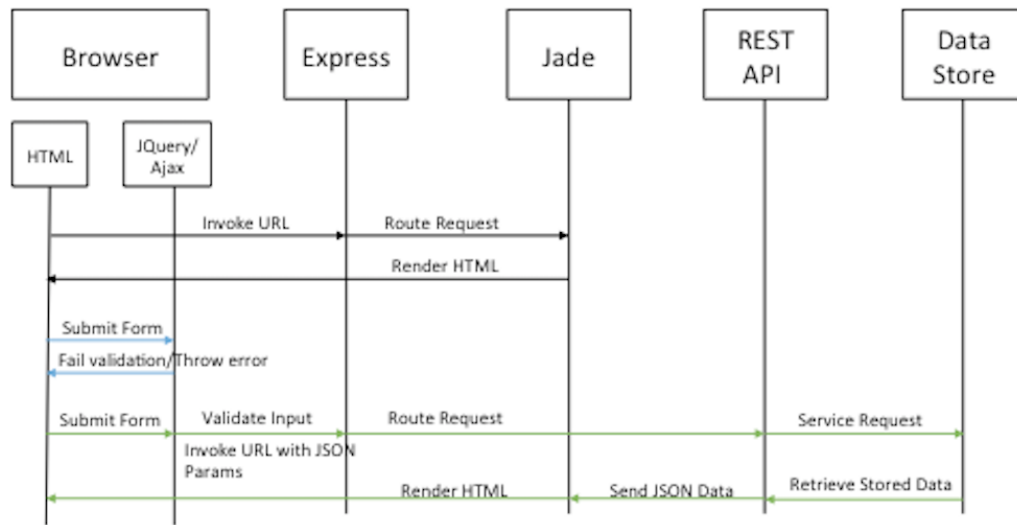


Figure 3.2: Request-Response Pipeline

The application has mainly two flows (Figure 3.2) that are described below.

1. **Static pages:** This flow is followed when serving static pages such as the landing page. In this case, the request from browser is intercepted by Express middleware which uses Jade template engine to render HTML

back to the browser.

2. **Dynamic Content:** Most of the functionality of the application is served via dynamic content. In this case, the user inputs are validated in the browser using jQuery/JavaScript. If the validation fails, user is given the error message. All this happens in the browser itself without a hop to the server. If the input validation succeeds, a JSON document is created by the client-side JavaScript. This document is posted to the back-end by a jQuery/Ajax call via HTTP verb (GET, PUT, POST, DELETE). The target is the REST web service hosted by Express middleware. Express redirects this request to the mapped JavaScript code which then services the request. This JavaScript code is responsible for rendering the response. Depending on functionality, it could make a call to the back-end data store, MongoDB in our case, to store or retrieve data (CRUD operations). The results from the store come in JSON format which are then interpolated by the JADE template engine to render the HTML back to the browser.

### 3.4 Summary of Technologies Used

The following table lists the libraries or frameworks used for building UToobe application.

Library/Module	Version	Summary
Body-Parser	1.14.0	Request body parser
Bootstrap	13.3.5	CSS and Component framework
Bootstrap-Fileinput	4.2.7	HTML5 file input for Bootstrap 3.x
Busboy-Body-Parser	0.0.9	Multipart form data parser
Express	4.13.3	HTTP Server
Express-Session	1.11.3	Session management library
Fluent-Ffmpeg	2.0.1	Media manipulation library
Gridfs-Stream	1.1.1	Interface to GridFS
Jade	1.11.0	Template engine
jQuery/Ajax	2.1.4	DOM manipulation & asynchronous JS
MongoDB	3.0.5	NoSQL data store
Morgan	1.6.1	Logger library
Node.js	0.12.7	Async, Non Blocking I/O framework
Nodemailer	1.10.0	Library for Sending Emails
NPM	2.12.1	Node Package Manager

Table 3.1: Summary and Version of Library/Frameworks Used

### 3.5 Data Model

To support the application the following four collections were used. The collections and their document instance are outlined below.

#### 3.5.1 Media Collection

Listing 3.1: Media Document

```

1 {  _id: Object,
2    title: String,
3    description: String,
4    keywords: String,
5    category:String,
6    fileName: String,
7    contentType:String,
8    likeCount: integer,

```



```

9      dislikeCount: integer,
10     userID: Object,
11     comment: Array,
12     fileId: Object,
13     createDate: Date }

```

Listing 3.1: Media Document

### 3.5.2 User Collection

```

1  {
2      _id: Object,
3      loginName: String,
4      passwd: String,
5      email: String,
6      firstName: String,
7      lastName: String,
8      dob: Date,
9      profilePic: Object,
10     mediaCount: integer,
11     commentCount: integer,
12     likeCount: integer,
13     dislikeCount: integer,
14     createDate: Date }

```

Listing 3.2: User Document

### 3.5.3 fs.chunks Collection

GridFS uses *fs.chunks* to store the media data.

```

1  { _id : Object,
2    filename : String,
3    contentType : String,
4    length : Double,
5    chunkSize : Double,
6    uploadDate : Date,
7    aliases : Array,
8    metadata : Object,
9    md5 : Object }

```

Listing 3.3: fs.chunk Document

### 3.5.4 fs.files Collection

GridFS uses *fs.files* to store metadata for the media.

```
1 {  _id : Object ,
2    files_id : Object ,
3    n : Double ,
4    data : Binary Data }
```

Listing 3.4: fs.file Document

## 3.6 Source Structure

This section describes the organization of the source code. The skeleton of the project was generated using express-generator [27] module. The project has the following structure:

- package.json - It is a manifest/configuration file that directs npm to install dependencies and modules.
- app.js - It is the starting point of the application. It loads required modules and starts the HTTP server.
- devDb\_Config.txt - contains database configuration details, namely, path to data store file, port, maximum number of connections and logging level.
- routes/ - contains Express application routing middleware.
- views/ - provides templates which are rendered and served by configured routes.
- config/ - contains application configuration files, for example, database config.

- public/ - contains all static files like images, styles and JavaScript.
- lib/ - contains common code and functionality shared by different parts of the project.

### 3.7 RESTful Services

The following table summarizes RESTful services used by **UToobe** application. The main parameters passed to these services are `uID` (userID) and `mID` (mediaID) which are the unique IDs of user and media respectively. HTTP verbs (GET, PUT, POST and DELETE) are used for performing read, update and delete operations.

Route	HTTP Verb	Description
/user/:uID	GET	Get user details
/user/:uID	PUT	Update a user
/user/:uID	DELETE	Disable a user
/user/signup	POST	Create new user
/media/:mID	GET	Get media details
/media/:mID	PUT	Update a media
/media/:mID	DELETE	Delete a media
/media/add	POST	Create new media
/media/recent	GET	Get most recent media
/media/mostLike	GET	Get most voted media
/media/search	POST	Get media list that match criteria
/media/list	GET	Get user media list
/media/clip/:mID	POST	Clip media
/media/manip/:mID	POST	Manipulate media

Table 3.2: List of RESTful Services consumed by **UToobe**

# Chapter 4

## Implementation

This chapter describes the implementation details of the application. I will go over the various use cases and how they were realized. These details are explained in terms of the screenshots, data-interchange formats, and the web-services [62] created for achieving the functionality. The application was developed using WebStorm [36] IDE [48]. The main technologies used in the application are jQuery [50], Ajax [39], Jade [14], Node.js, NPM [57] (Node Package Manager), MongoDB [56] and MongoDB Node.js Driver [30].

### 4.1 Navigation Bar

A navigation bar is the navigation header placed at the top of every page after the user has logged in. It provides the ability to navigate to various parts of the application. I used Bootstrap *navbar* and *navbar-default* class with HTML5 *nav* tag. The navigation bar can extend or collapse, depending on the screen size.

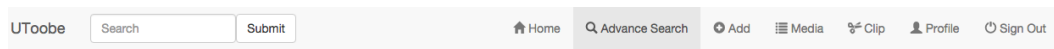


Figure 4.1: Navigation Bar

As can be seen in the Figure 4.1, Bootstrap components called Glyphicon have been used. Bootstrap includes over 250 free glyphs in font format from the Glyphicon Halflings set. The navigation bar is added to the top of every page by use of Jade *includes* API.

```
1 <ul class="nav navbar-default">
2   <li><a href="/media/add"><span class="glyphicon
   glyphicon-plus-sign">
3   </span> Add</a>
4 </li></ul>
```

Listing 4.1: Navigation bar with (+) Glyphicon

## 4.2 Authentication

The following screenshot (Figure 4.2) shows the landing page of the application. This page provides a new user with the ability to create an account while an existing user can use this page to enter his/her credentials and login to the system.

## Welcome to UToobe! Please sign In

Login Name

Password

☒ Remember me

Sign In

[Forgot your password?](#)

New user? [Create new account](#)

Figure 4.2: Login Page

The validation script (loginCtrl.js) together with a call to HTTP POST to `/user/login` ensures that a user is valid. If a user credentials are not valid then an error message is thrown on the Login Page and the user is prompted to try logging in again. On successful authentication, a user is taken to the dashboard page. I used a simple key value comparison for authentication. Express-session module has been used for storing the user login name and userID. If a user has forgotten his/her password, they can select *Forgot your password?* link and a Bootstrap modal box pops up where they can enter their email address. The current password would then be emailed to them. Nodemailer [32] module is used to send emails.

jQuery Ajax is used for posting the JSON document containing user-

name and password to /user/login.

```
1 $.ajax(  
2     {  
3         type: 'POST',  
4         data: user,  
5         url: '/user/login',  
6         dataType: 'JSON'  
7     }
```

Listing 4.2: Authentication request

The following callback function ensures that the userID exists in the data store and the supplied password is valid. Express-session [26] is used for storing session details.

```
1 userMgr.authUser(req.body, function validateUser(err,  
2     result) {  
3     ..... //query db  
4     if (result !== null) {  
5         //set user details in the session  
6         req.session.userID = result._id;  
7         req.session.loginName = result.loginName;  
8     }  
9     resp.send((result === null) ? {'msg': 'User does  
    not exist'} : {'msg': 'SUCCESS'});  
10 });
```

Listing 4.3: Credentials Validation callback

The following call to MongoDB tries to find a user document that contains matching username and password.

```
1 collection.findOne(data, callback);
```

Listing 4.4: Data Store findOne callback

### 4.3 Account Creation

This use case handles creation of a new user in the system. Users can navigate to this screen by selecting *Create new account* link on the Login Page.

## Sign Up

Please fill out the details to register.

Birth Date

☒ I agree with the [Terms and Conditions](#).

Create my account

Figure 4.3: Account Creation Page

It takes the following inputs in the HTML form - username, password,



loginID and date of birth to create a unique user in the system. On filling the details and submitting the form, the form validation takes place on the client side using *signUpCtrl.js*. On a successful validation, a JSON document is created containing user details. jQuery-Ajax is used to call HTTP PUT to URI - `/user/signup`. Express then routes the request to the associated callback methods. The following jQuery-Ajax called is made to create a new user in the system.

```
1 $.ajax(  
2     {  
3         type: 'PUT',  
4         data: newUser,  
5         url: '/user/signup',  
6         dataType: 'JSON'  
7     }  
8 );
```

Listing 4.5: Create new user request

The following associated callback function is used for forwarding JSON document to the URI.

```
1 router.put('/signup', function createUser(req, res) {  
2     userMgr.addUser(req.body, function insertCb(err, result) {  
3         .....  
4         res.send((err === null) ? {msg: 'SUCCESS'} : {  
5             msg: err});  
6     });  
7 });
```

Listing 4.6: Create User callback

The following MongoDB JS function is called to insert the user JSON document in the data store. Refer to Listing 4.7.

```

1 {
2   "_id" : ObjectId("561ac7xxxxxx"),
3   "loginName" : "johnK",
4   "passwd" : "123secret",
5   "email" : "john@macworld.com",
6   "firstName" : "John",
7   "lastName" : "King",
8   "dob" : ISODate("1975-11-26T00:00:00Z"),
9   "mediaCount" : 0,
10  "commentCount" : 0,
11  "likeCount" : 0,
12  "dislikeCount" : 0,
13  "createDate" : ISODate("2015-10-11T20:33:09.206Z")
14 }
15 collection.insert(data,callback)

```

Listing 4.7: User Data (JSON) and insert callback

## 4.4 Dashboard

On successful login, a user is taken to dashboard page. This page consists of two sections with thumbnails view of shared media. Top section displays the media sorted by popularity while the bottom section displays the most recently uploaded media. Selecting any of the media files takes the user to the View media.

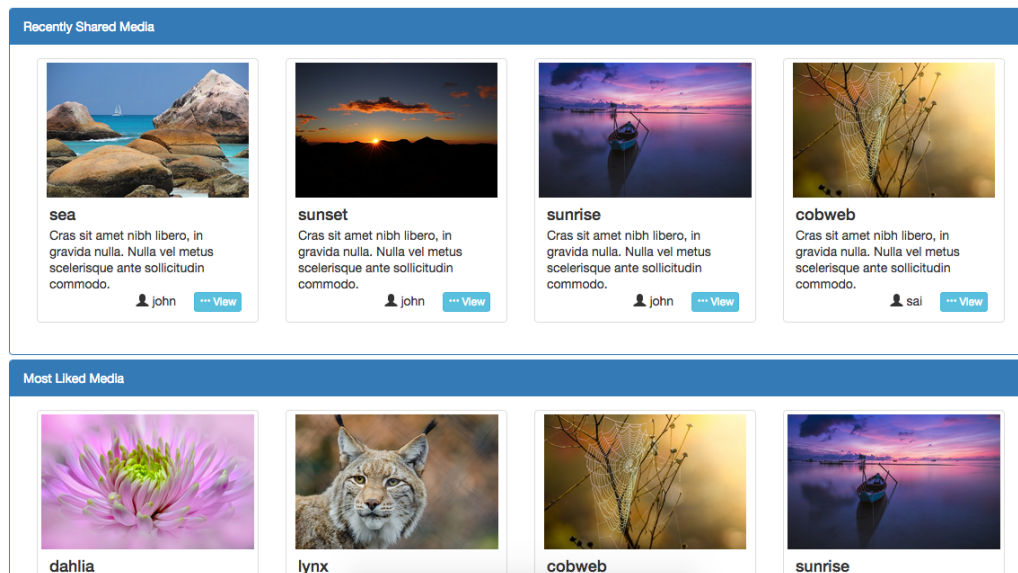


Figure 4.4: Dashboard Page

The page uses Jade template engine on the server side to render the content. This is a two step process. First `GET /media/dashboard` URI is invoked which retrieve a static content. After that two separate jQuery Ajax calls are made to `GET /media/recent` and `/media/mostLiked`. The result is an array of JSON documents which is parsed by the Jade engine.

```

1  a) router.get('/recent', function getRecentList(req,
    res) {
2      .....;
3      res.json(rSet);
4  });
5  b) router.get('/mostLikes', function getMostLike(req,
    res) {
6      .....;
7      res.json(rSet);
8  });

```

Listing 4.8: REST calls for get Recent and Most Likes

```

1  /**** MongoDB JS data store find function : ****/
2  exports.getRecent = function (limit, cb) {
3      var options = {
4          "limit": limit,
5          "sort": [['createDate', 'desc']],
6      }
7      return connMgr.mediaColl().find({}, options).
          toArray(cb);
8  };

```

Listing 4.9: DB call for get Recent Media

## 4.5 View Media

On selecting *View* on thumbnail of the media in the Dashboard, the application takes the user to View media. On this page the user can play the media if it is a video file or simply see a larger version in case of static images. User can also add his or her comments, vote for the media, and read other users comments.

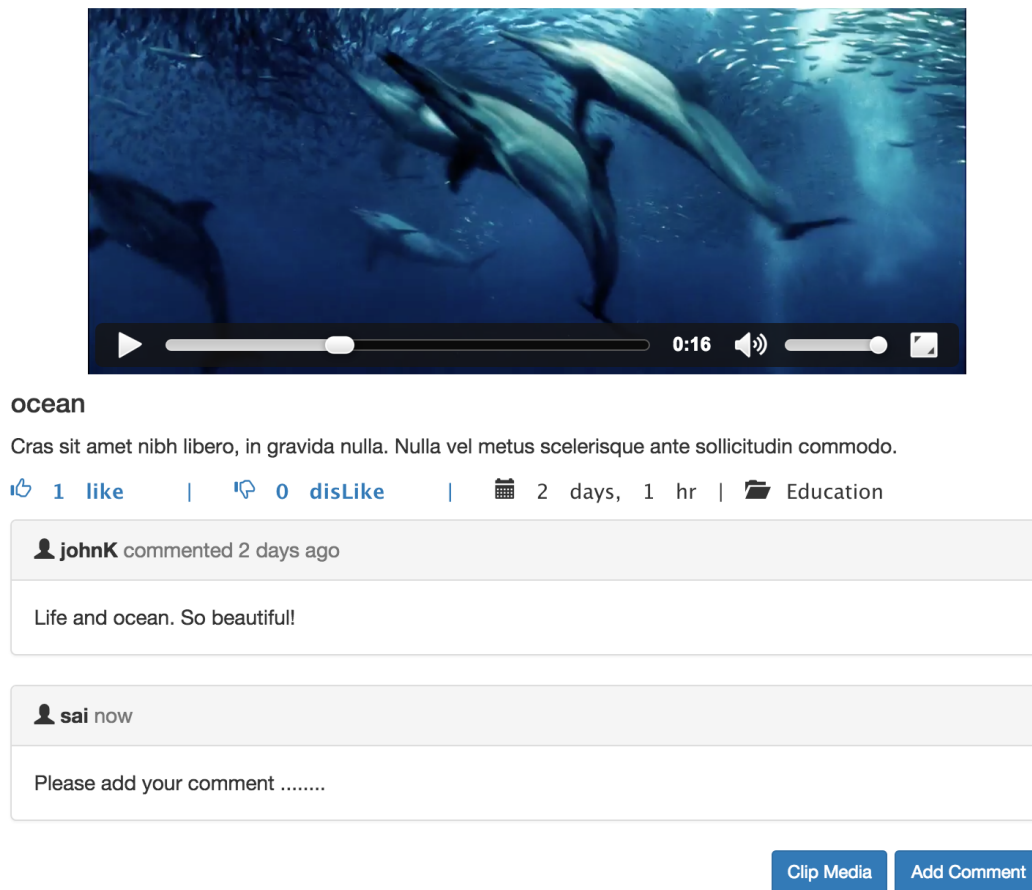


Figure 4.5: Comments on Media

The view is composed of the Jade template and call to HTTP GET `/media/:mID/view`. The request returns media document associated with given *mID*. On adding a comment and pressing the *Add Comment*, an HTTP POST `/media/:mID`.

```

1 mediaMgr.upateMedia(req.params.mID, function (err,
    result) {
2     res.render('mediaView',result);
3 });
4 /**** MongoDB data store find and update ****/
5 collection.findOne(data,callback)
6 collection.update({_id:"foo"}, data, callback)

```

Listing 4.10: Add Comments

## 4.6 Search

The application provides the ability to search for media by title, keywords, author and category. The search page can be reached by clicking *Advanced Search* on the navigation bar. On entering the value in the search form and hitting submit, jQuery extracts the values of the form and a JSON document is created. jQuery Ajax posts the form data to `/media/search`. The result is a list of documents that match the search criteria. Array of JSON documents is parsed, html constructed and then the content is injected into the page.

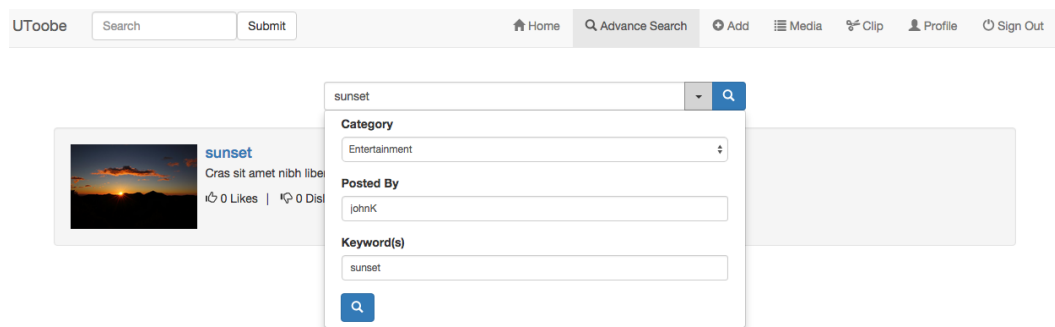


Figure 4.6: Search Capability

```

1 searchDoc = {
2     'title': searchTitle,
3     'category': searchCat };
4 $.ajax(
5     {    type: POST,
6         data: searchDoc,
7         url: /media/search,
8         dataType: JSON })

```

Listing 4.11: Search Request

The following callback function is called to get a list of media that matches the search criteria.

```

1 router.post('/search', function getSearchResult(req,
2     res) {
3     mediaMgr.findMedia(req.body, function (err, result){
4         res.json(result); } })
5     /** Search results injected usint jQuery */
6     $('#resultSet').html(htmlLine);

```

Listing 4.12: Search results (JSON) array

The following MongoDB JS function is called to find the matching media:

```

1 collection().find(data).toArray(callback);
2 /** regular expression to match search criteria */
3 collection().find( { title: { $regex: result[,
4     $options: 'i'] } } );

```

Listing 4.13: Data Store find call

Async.js library is used with waterfall pattern along with async.each to retrieve data in two collections on the common key - userID. The media document contains the userID but the UI displays the loginId associated with user name. This could have been done with two calls but it was more efficient this way to do as one call on the server side.

```

1  /**Use Async library to control execution of multiple
   callbacks, use result of one call back in the
   following callback function */
2  async.waterfall([
3      function (cb) {
4          // find matching media documents
5          mediaMgr.findMedia(req.body, function (err
6              , result) { cb(err, result); });
7      },
8      function (result, cb) {
9          // calculate and set elapsed time
10         var currDt = new Date();
11         result.forEach(function (r) {
12             r.timeElapsed = helper.getTimeDiffStr(
13                 r.createDate);
14             r.commentCount = r.comments.length;
15         });
16         cb(null, result); } ,
17     function (result, cb) {
18         // parallel exec to get loginId for userId
19         async.each(result, function (rec, cb) {
20             userMgr.getLoginName(rec.userId,
21                 function (err, loginInfo) {
22                     rec.loginName = loginInfo.
23                         loginName;
24                     cb(); });
25         }, function (err) {
26             if (err) { // if one of the iterations
27                 produced an error, all processing
28                 will stop.
29                 console.log('err has occurred');
30             } else {
31                 console.log('Records successfully
32                     updated');
33                 cb(null, result); } }); }
34 ], function (err, rSet) { // return resultSet
35     res.json(rSet);
36 });

```

Listing 4.14: Async Waterfall to fetch data from Multiple collections



## 4.7 Add Media

UToobe provides the ability to upload and share media. The Add Media page can be reached by clicking *+ Add* on the navigation bar. The web page consists of HTML form that prompts user to provide details for the media to upload - title, keywords, description and category. A user can then select the file to be uploaded either by browsing the local file system or using a drag and drop. User can also remove the file for upload from this preview box by selecting remove. The upload-view provides the ability to preview the uploaded file i.e. view image, play video, audio etc.

This form template uses bootstrap-fileinput plug-in. This plug-in is built to work with Bootstrap CSS3 styles. It allows developers to setup file picker directly through html tags or using jQuery. I have used the former. Snippet of Jade plug in code is shown in Listing 4.15.

```
1  /*** Jade head ***/
2  link(href= '/css/fileinput.min.css', media= 'all', rel= '
    stylesheet', type= 'text/css')
3  script(src= '/bootstrap-fileinput/js/plugins/
    canvas-to-blob.min.js', type= 'text/javascript')
4  script(src= '/bootstrap-fileinput/js/fileinput.min.js',
    type= 'text/javascript')
5  /*** Jade body ***/
6  input#fileSelectedData.file.input-sm(type= 'file',
    data-show-upload= 'false')
```

Listing 4.15: bootstrap-fileinput plug-in

On submitting the form, the validation for the form fields takes place by *addMediaCtrl.js* followed by creation of media JSON object. jQuery Ajax call is made to HTTP PUT */media/add*. The multipart request form data is

**Description**

Gras sit amet nibh libero, in gravida nulla. Nulla vel metus scelerisque ante sollicitudin commodo.

**Keywords**

leopard

**Category**


Misc

**Browse to select**


Selected file can be Previewed/Cancelled



snow-leopard-1006542\_\_340....

 snow-leopard-1006542\_\_340.jpg

 Remove

 Browse ...

Add Media

Figure 4.7: Add Media Page

handled by busboy-body-parser module followed by callback to uploadMedia. The uploaded file data and its attributes are passed to data store for persistence. The gridfs-stream module instance is used for creating writestream to persist data in MongoDB store. The media attributes below are stored in the mediaCollection. Listing 4.16 code extracts the form data and assembles it to create a media JSON document.

```
1 var formData = new FormData();
2   var newMedia = {
3     'title': $('#textTitle').val(),
4     'description': $('#textAreaDesc').val(),
5     'keywords': $('#textKeyword').val(),
6     'category': $('#optCategory').val()
7   };
8   formData.append("newMedia", JSON.stringify(
9     newMedia));
10  formData.append("fileSelectedData", $('#fileSelectedData')[0].files[0]);
```

Listing 4.16: jQuery extract form data

The following code calls HTTP PUT to create a new media object in data store.

```
1 $.ajax( { type: 'PUT',
2           data: formData,
3           cache: false,
4           contentType: false,
5           processData: false,
6           url: '/media/add' })
```

Listing 4.17: Create Media Request

The following callback function is used to parse the multipart request data and then upload media data and attributes to data store.

```

1 router.put('/add', busboyBodyParser({limit: '100mb'}),
    function uploadMedia(req, res) {....}
2 // use gridfs-stream instance to create a write Stream
3 var writeStream = connMgr.getGFSInstance().
    createWriteStream({
4     _id: imageStorageID,
5     filename: mediaData.name,
6     mode: 'w',
7     content_type: mediaData.mimetype
8 });
9 writeStream.write(mediaData.data);

```

Listing 4.18: Multipart request parser with Write request

The following displays the structure of Media JSON.

```

1 {  "_id" : ObjectId("56217xxxxx"),
2    "fileName" : "oceans.mp4",
3    "contentType" : "video/mp4",
4    "title" : "ocean video",
5    "description" : "View of the ocean",
6    "keywords" : "ocean birds",
7    "category" : "Entertainment",
8    "comments" : [
9      {  "userID" : ObjectId("5621xxxxxxx"),
10         "createDate" : ISODate("2015-10-16T22
11           :16:25.021Z"),
12         comment: "I like this video. Very
13           refreshing" }
14    ],
15    "likeCount" : 0,
16    "dislikeCount" : 0,
17    "fileID" : ObjectId("562177398xxxxx"),
18    "userID" : ObjectId("561ac7855xxxxx"),
19    "createDate" : ISODate("2015-10-16T22:16:25.021Z")
20  }
21 /**** MongoDB insert function : ****/
22 collection().insert(data, cb)

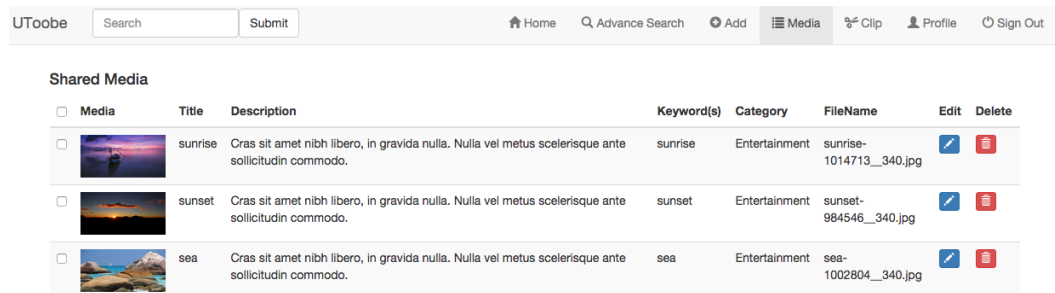
```

Listing 4.19: Media data persistence in Data Store

## 4.8 My Media

This functionality queries the data store and renders a list of the media uploaded by a user. An HTTP GET `/media/list` call is made and `userId` is passed to data store to query for media(s) uploaded by user. The view is made up of Jade template and iterator is used to parse the JSON document array. The list-view also provides the following:

- The ability to edit media details like title, description, keywords, and category by using HTTP POST `/media/:mID`.
- The ability to remove the media from the store by calling HTTP DELETE `/media/:mID`.



<input type="checkbox"/> Media	Title	Description	Keyword(s)	Category	FileName	Edit	Delete
<input type="checkbox"/>	sunrise	Cras sit amet nibh libero, in gravida nulla. Nulla vel metus scelerisque ante sollicitudin commodo.	sunrise	Entertainment	sunrise-1014713__340.jpg		
<input type="checkbox"/>	sunset	Cras sit amet nibh libero, in gravida nulla. Nulla vel metus scelerisque ante sollicitudin commodo.	sunset	Entertainment	sunset-984546__340.jpg		
<input type="checkbox"/>	sea	Cras sit amet nibh libero, in gravida nulla. Nulla vel metus scelerisque ante sollicitudin commodo.	sea	Entertainment	sea-1002804__340.jpg		

Figure 4.8: My Media List

```
1 router.get('/list', function getMediaList(req, res) {
2   mediaMgr.getUploadedList(req.session.userId,
3     function (err, result) {
4       // setting result in resultSet for Jade
5       iterator.
6       res.render("mediaList", {resultSet: result}
7     );
8   });
9 });
```

Listing 4.20: Jade Iterator for parsing document array

```

1  /**** MongoDB data store find, update, remove ****/
2  collection.find({_id:"foo"}.toArray (callback))
3  collection.update({_id:"foo"}, data, callback)
4  collection.remove({_id:"foo"}, callback)

```

Listing 4.21: Datastore find, update, remove

## 4.9 Clip

This use case allows the user to manipulate the media. A user can upload a media file or manipulate a shared media. Media manipulation includes horizontal and vertical flip option for images and clipping for a video. A user can then choose to save the clipped or manipulated media.



snow leopard

Snow leopards are slightly smaller than the other big cats but, like them, exhibit a range of sizes, generally weighing between 27 and 55 kg (60 and 121 lb), with an occasional large male reaching 75 kg (165 lb) and small female of under 25 kg (55 lb). They have a relatively short body, measuring in length from the head to the base of the tail 75 to 150 cm (30 to 60 in). However, the tail is quite long, at 80 to 100 cm (31 to 39 in), with only the domestic-cat-sized marbled cat being relatively longer-tailed. They are stocky and short-legged big cats, standing about 60 cm (24 in) at the shoulder.

Horizontal Flip

Vertical Flip

Save Media

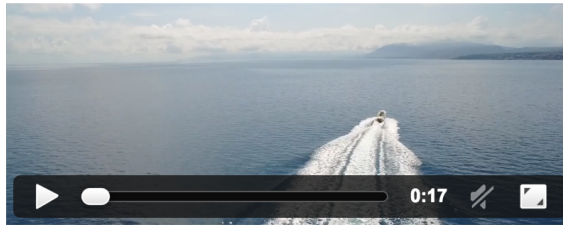
Figure 4.9: Media Manipulation

Fluent-ffmpeg module is used to perform media manipulation. For instance on selecting the *vertical flip* option available to images, a call to HTTP

PUT is made to `/media/manip/:mID` with details of the media and required manipulation request. Results of media manipulation are then rendered to the client. A user then has the option to save the media.

```
1 router.post("/manip/:mID", function manMedia(req, res)
2 {
3     ....// use ffprobe to get the format of the
        stream
4     ....//vertically flip the media
5     ffmpeg(r_stream)
6         // format of stream
7         .format('mjpeg')
8         .outputOptions('-vf vflip')
9         .output(w_stream) ;
10 });
```

Listing 4.22: Vertical flip Request



### Ocean View

It goes without saying that Australia is generally perceived as a distant country - we can't deny geography. The way we see it? A diverse country awaiting to be discovered. There are more than a thousand reasons to head straight to Australia and experience something different. On this occasion, we spoke with Verity Nelson about Western's Australia Kimberley region

Duration:  Start Pos:

[Clip Media](#) [Save](#)

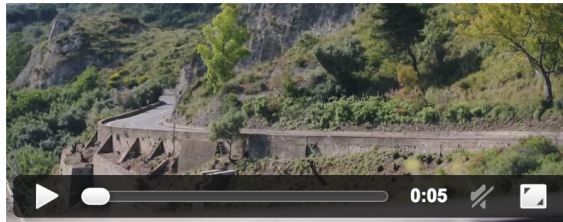


Figure 4.10: Media Clip

In Figure 4.10, on selecting *clip media* option available to videos, a call to HTTP PUT is made to `/media/clip/:mID` with details of the media and required manipulation params, namely, duration and start position.

```
1 router.post("/clip/:mID", function manMedia(req, res)
2 {
3     ....// use ffprobe to get format of stream
4     ffmpeg(r_stream)
5         .setStartTime('00:00:10')
6         .duration(5)
7         .save(w_stream);
8 });
```

Listing 4.23: Clip Media Request



## 4.10 Profile

This use case provides the ability to view the profile of the currently logged in user. Profile details consists of name, loginID, current number of like(s) and dislike(s) votes, number of uploaded media, user creation date and number of comments added by user. This functionality achieved by a call to HTTP GET `/user/profile`. UserID stored in the session is used for finding the associated user details. Jade template is used for building the page and Jade interpolation is used for displaying the details of the user.

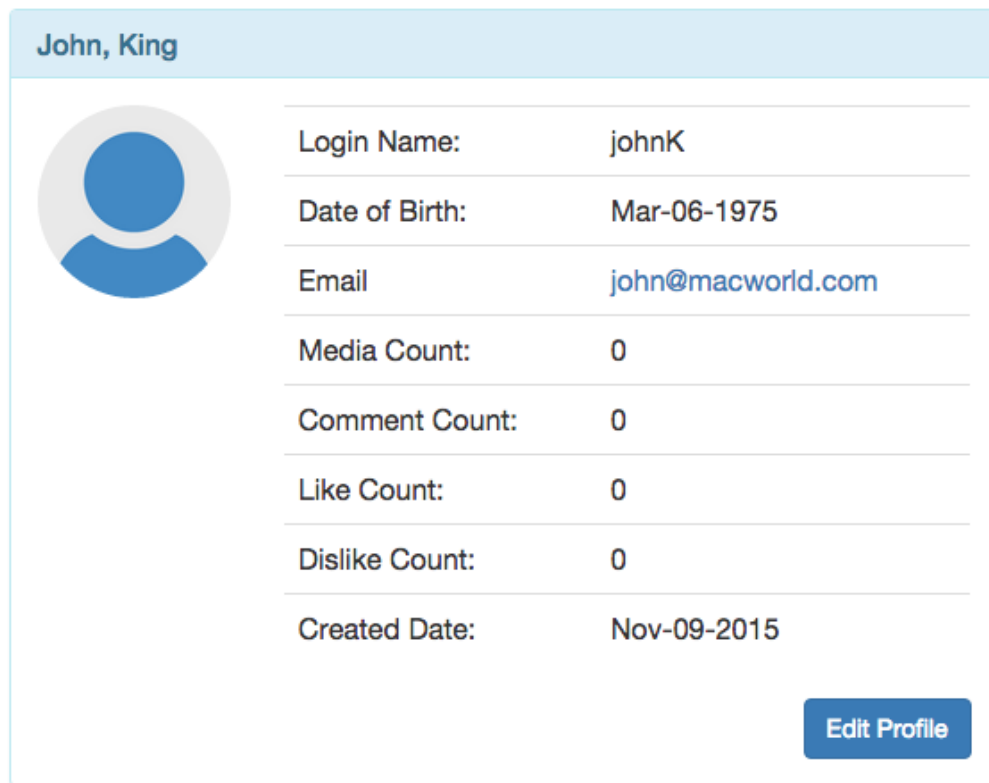


Figure 4.11: User Profile

The profile view also provides the ability to edit details of the profile - name, email and date of birth. This is achieved by a call to HTTP PUT `/user/:userID`.

```
1 router.get('/profile', function viewProfile(req, res)
2   {
3     userMgr.getUser(req.session.userID,
4       function getProf(err, result) {
5         result.createDate = helperFn.
6           convertDtToString(result.createDate);
7         result.dob = helperFn.convertDtToString(
8           result.dob);
9         res.render('userProfile', result);
10      });
11  });
12 /**** MongoDB JS data store find and update : ***/
13 userColl().findOne({_id: "foo"}, cb);
14 userColl().update({_id: "foo"}, {$set: userDoc})
```

Listing 4.24: View Profile callback

## 4.11 Sign Out

Sign Out link in the navigation bar provides the functionality of destroying the user session and rendering a logout page. This page provides the ability to re-login.

```
1 router.get('/logout', function logOut(req, res, next)
2   {
3     req.session.destroy();
4     res.render("logout");
5   });
```

Listing 4.25: LogOut callback

## 4.12 Utilities and Configuration Files

MongoDB configurations are stored in `db_config.js`. This configuration consists of the name of the database instance and names of the collections. On application server startup, this configuration is read by *connMgr.js*. It then creates a pool of database connections and make them available for servicing database requests.

```
1 exports.config = {  
2   db_params: {  
3     host: "localhost",  
4     port: 27017,  
5     poolSize: 6,  
6     dbName: "utoobe"  
7   },  
8   userColl: "users",  
9   mediaColl: "medias"
```

Listing 4.26: Database Configuration

## Chapter 5

### Conclusion

UToobe application building was a valuable exercise in my goal to learn new tools and emerging web technologies. This chapter documents the lessons learnt and enhancements that I would like to make in future.

#### 5.1 Lessons & Reflections: An Introspection

##### 5.1.1 JavaScript

In my professional career I have mostly worked with Java. I have used JavaScript in the past, but mostly for the front-end development. In this application, I have used JavaScript both on front-end as well as the server side. The code in JavaScript is very concise and compact. Amount of code written for this application is approximately 1500 lines. This includes front-end code (Jade, JavaScript), as well as back-end (Express, Node.js, etc.). In my estimate, the same thing would have taken a plethora of classes and over three times more lines of code using Java technologies [13]. For example, in order to parse a JSON message in a Java program, I would have to use libraries like Gson which would convert JSON message into Java classes. After that I would have to traverse the Java classes to extract the data. In contrast

JavaScript allows me to traverse JSON strings directly and extract the data that I need. This inbuilt support for JSON reduces the size of code and makes it lucid.

### 5.1.2 Node.js

Software development in Node.js was a complete paradigm shift, especially due to my prior experience with Java. Initially it was very difficult to understand and develop code using Node.js's single-thread model and the concept of non-blocking I/O. But it had a small learning curve and soon I was comfortable with it. I would strongly recommend Node.js as a powerful tool for any web developer for the following reasons:

**Single thread model:** I had faced performance issues with Java programs even when using multi-threading. I appreciated how Node.js takes care of performance by using a simple single-threaded design thereby removing the overhead of thread context switching, and resource hogging by each thread. The single thread model frees the developers from the complexity of writing multi-threaded code for performance, as is done in Java [16]. The latter requires developers to hone difficult and bug-prone concepts like thread-pools, class/instance level locks and synchronization, inter-thread communication, and deal with issues like race-conditions, thread-starving, runaway threads, etc. At the same time, Node.js does allow you to make use of multiple cores of your machine by allowing you to spawn child processes or send messages to additional worker processes. In this design, one thread manages the flow of

events and other threads perform system calls.

However, this simplicity in runtime design comes at the cost of code complexity. Callbacks were hard to debug. Order of execution of the callbacks is not predictable. I would suggest keeping nested callbacks to a minimum to reduce complexity of debugging and error-handling. Also naming functions rather than using anonymous functions helped. Use of Async.js `bind()` to attach a context to the object can make Async callback patterns easy to follow.

**Non Blocking I/O:** The single thread model of Node.js requires that all I/O operations are non-blocking, so that the main thread is never waiting for I/O. This is realized using callbacks functions. Requiring I/O operations to be non-blocking, and handling them via separate pool of threads is a clever way to improve CPU utilization.

**Succinct:** As compared to Java, the code written using Node.js is very compact and succinct. Dynamic typing helps to write more loosely coupled applications and easier integration. Closures help scoping the access of the function's variables and this in turn helps in creating a concise and expressive code.

### 5.1.3 jQuery-Ajax and Jade

AngularJS is the *A* in the MEAN stack and quite popular. AngularJS is a powerful two-way data-binding framework that allows usage of directives for creating reusable custom components. It is [40] built for SPA(Single Page Applications) [60] and offers capabilities much more than DOM manipulation.

However, it has a high learning curve. I spent some time understanding its view patterns but later decided in favor of jQuery-Ajax. I would like to learn it in future for creating dynamic SPA where a page is highly interactive.

jQuery [50] library worked very well for making Ajax [39] requests, manipulating elements, triggering and listening for events, selecting elements from the DOM, getting and setting form input values and traversing the DOM. This complemented Jade template engine and acted as a facade to standardize and ease the task of interacting with elements on a web page.

Jade provides the features of *interpolation*, *iterator*, *inheritance*, *includes* and embedded logic to substitute values in generated HTML. One can write html in few lines of Jade code. It does not required closing tags but is space and indentation sensitive. This can be annoying at first. I used tools like Jade2HTML and HTML2Jade to help debug the issues.

If this application was developed using Java web technologies, this part of the application would have been implemented using Servlets or JSPs. There is a tradeoff between the two technologies. Although the statically typed Java framework makes it easy to maintain/refactor code [19] and gives better performance for high volumes, the write-compile-deploy cycle slows down development. The dynamically typed frameworks like jQuery/Jade provide for faster prototyping and increased developer productivity.

#### 5.1.4 Bootstrap

As a server side engineer, I have always been uncomfortable venturing into the creative world of UI development. I have found the use of CSS, view controls and pixel controls to be challenging. The availability of the frameworks like Bootstrap with inbuilt components and styling capabilities helped put together a user-interface that looks professional rather than the work of a rookie web-developer. The responsive UI with the inbuilt grid column control was very powerful. I did not get a chance to use it to its full potential but would like to do so in future.

#### 5.1.5 Express

Use of Express as a HTTP server turned out to be very simple. It received the web-requests and carried out the functionality of routing, redirection and caching. Configuring Express was much simpler than a typical HTTP server which requires configuring web.xml. Express performs routing using GET, POST, PUT, DELETE methods which take route path parameters and invoke the mapped handlers at runtime. As a route can have one or more handler functions, this renders flexibility, enables *handler chaining* and facilitates code reuse. The routing configurations also provides the ability to accept optional parameters and match wildcards.



### 5.1.6 MongoDB

This was my first interaction with a NoSQL database. Having worked with RDBMS all my professional life, I was able to appreciate the ability of MongoDB to handle unstructured data. RDBMS systems become slow as the amount of data increases. They are not able to scale horizontally by utilizing multiple servers, but mostly scale vertically by demanding more capacity from a single server. Though lately some of the RDBMS systems have come up the feature of *sharding* (ability to distribute data partitions across multiple servers), it however, does not come out of the box. MongoDB on the other hand is capable of providing horizontal scaling by distributing the data across multiple servers [55].

Dealing with large-sized data pieces is also very slow as well as cumbersome in RDBMS system. In the RDBMS world, they are referred as LOBs (Large Objects) which can be of type BLOB or CLOB [20] (binary or character data respectively). RDBMS systems store the entire LOB at a single location. This makes storage and retrieval operations very slow. MongoDB on the other hand, via its GridFs interface [54], divides large files into smaller chunks of default size 255 kilobytes, and stores each chunk separately. This not only speeds up read/write operations, but also provides the ability to load only certain parts of the file instead of the whole. What makes it even better is that all this happens behind the scene and developers do not have to deal with individual chunks. Developer's code stores and retrieves the full file and chunking happens transparently.

Another aspect of MongoDB is that it provides an easy, JSON oriented interface for storing large, unstructured data. This was especially helpful since I used full-stack JavaScript which has native support for JSON.

#### 5.1.7 Development Environment

**IDE and Supervisor:** I used WebStorm [36] from JetBrains for the application development. It was lightweight and provided a good set of debugging tools. Using Supervisor programs like *nodemon* [8] helped speed up development. In the development environment, Nodemon automatically restarted the application whenever it detected any modification in the source code or if the application crashed.

**NPM:** This is a packaging tool that comes with Node.js and greatly speeds up development. The application dependency declaration file *package.json* is a simple JSON file. Developers can share the libraries easily by pushing them to the npm registry.

Packages are constantly being updated on Node.js platform. There is a big community of developers enhancing and modifying the platform. Therefore, restrict the module versions in your manifest file i.e. *package.json* to use *fixed* versions rather than *~*(tilde) or *^*(caret) before a module version. The *~* matches the most recent minor version and *^* forces package-manager to update to the most recent major version. Use *node update* command to view if you have any latest updates in the npm registry and then perform the update. This will help the builds from failing.

### 5.1.8 Documentation

As Node.js is evolving rapidly and the community is growing, the documentation is not able to keep up the new code and changes. Node.js foundation is doing a good job but it is not enough. Stack Overflow [35] Q & A and developer blogs helped fill that void.

### 5.1.9 Performance

File Size	Type	Time to Upload (milliseconds)
6.5 MB	Image File	143
11.5 MB	Audio File	231
23 MB	Movie File	662

Table 5.1: Media Upload Time(milliseconds)

I found the performance of the application to be satisfactory. Uploading a 23 MB file takes approximately 662 milliseconds. A more formal performance valuation would require the application to be hosted on the net, together with a larger size test data.

### 5.1.10 Testing

Testing of application developed using MEAN stack can be performed with tools like Mocha, NodeUnit, Jasmine etc. For stub generation, tools like JsMockito, Sinon.JS etc. can be used. The testing follows the same patterns as in Java technologies. I did not get time to write formal test scripts. This would be an enhancement for the future.

## 5.2 Source Code

The source code and documentation for **UToobe** is available in a public Git repository at BitBucket.com. The following links can be used to access it:

`https://vgambhir@bitbucket.org/vgambhir/utoobe.git`.

A demonstration of the application can be found at:

`https://vgambhir@bitbucket.org/vgambhir/utoobe/demo/utoobeDemo.swf`

## 5.3 Future Work

Given more time, I would have liked to incorporate more features into this application, ability to rank users based on popularity of the media posted by them, upload profile picture, etc. I would also like to further explore ffmpeg [4] library and provide other interesting abilities to edit the media.

Passport [10] module is an authentication middleware that supports a comprehensive set of strategies to support authentication against Facebook, Twitter and more. I would like to open the application to use this broader authentication.

Mongoose [6] is an object modeling package for Node.js. It works like an ORM and provides schema-based solution to model the application data. It includes built-in type casting, validation, query building and provides hooks for business logic. I would like to use this in place of MongoDB Node.js native driver. For a small application the native driver sufficed.

I did not make use of AngularJS [40] due to lack of time. AngularJS is part of the MEAN stack and is very popular. In future enhancements to this application I would like to explore AngularJS.

## 5.4 Summary

All the components used in this application worked very well with each other. Each component provides a particular functionality of the stack. jQuery provides client-side validation and creation of HTTP request. Express provides the HTTP server functionality and routes requests to appropriate code. MongoDB provides the persistent store with a JSON interface.

Node.js provided an overall framework with a single thread model and a non-blocking I/O. Availability of libraries like busboy-body-parser, fluent-ffmpeg, express-session, morgan, gridfs-stream and MongoDB reduced the application development effort drastically.

It was very enlightening to learn NoSQL databases, and it gave me a new perspective on storing unstructured and large data efficiently. The programming interface as well as their capabilities are very different from the traditional RDBMS systems.

This application helped me to learn the latest web-technologies and appreciate their strengths better. These new technologies provide an easy way to quickly build an application and greatly reduce the time to market, since they have a lot of inbuilt functionality. Developers can concentrate on business

functionality rather than spend too much time and effort on the peripheral tasks. At the same time, there are so many new technologies and acronyms floating around that it becomes very confusing to a rookie. I was able to parse the various technologies and understand each one's role in the solution stack. As such the goal of this project was successfully achieved.

## Bibliography

- [1] Zing Design. Top 10 web development trends and predictions for 2015.  
<http://www.zingdesign.com/top-10-web-development-trends-and-predictions-for-2015/>, November 2015.
- [2] BSON Doc. BSON Document. <http://bsonspec.org/>, November 2015.
- [3] Express Doc. Express Doc. <http://expressjs.com/>, November 2015.
- [4] FFMpeg Doc. FFmpeg public site. <http://www.ffmpeg.org/>, November 2015.
- [5] JSON Doc. JSON Document. <http://www.json.org/>, November 2015.
- [6] Mongoose Doc. Mongoose Object Modeling. <http://mongoosejs.com/>, November 2015.
- [7] Node.js Doc. Node.js Site. <http://nodejs.org/en/>, November 2015.
- [8] Nodemon Doc. Nodemon Site. <http://nodemon.io/>, November 2015.
- [9] NoSQL Doc. NoSQL. <http://nosql-database.org/>, November 2015.
- [10] Passport Doc. Passport Site. <http://passportjs.org/>, November 2015.

- [11] Oracle Docs. Large Object in Database.  
[http://docs.oracle.com/cd/B28359\\_01/appdev.111/b28427/pc\\_16lob.htm#i1003528/](http://docs.oracle.com/cd/B28359_01/appdev.111/b28427/pc_16lob.htm#i1003528/),  
November 2015.
- [12] Jon Duckett. *JavaScript and JQuery: Interactive Front-End Web Development*. Wiley, 2014.
- [13] Ralph Johnson John Vlissides Erich Gamma, Richard Helm. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 2014.
- [14] Jade Home. Jade home. <http://jade-lang.com/>, November 2015.
- [15] David Hows and Peter Membrey. *MongoDB Basics*. Apress, 2015.
- [16] Paul Hyde. *Java Thread Programming*. Sams Publishing, 2014.
- [17] Sean Lang. *Web Development with Jade*. packt publishing, 2014.
- [18] Azat Mardan. *Practical Node.js: Building Real-World Scalable Web Apps*. Apress, 2014.
- [19] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.
- [20] Jon Paris. BLOB - CLOBS.  
<http://www.ibmssystemsmag.com/ibmi/developer/general/BLOBs,-CLOBs-and-RPG/>, November  
2015.



- [21] Mithun Satheesh. *Web Development with MongoDB and NodeJS* . Packt Publishing, 2015.
- [22] Open Source. Async JS library. <http://github.com/caolan/async/>, November 2015.
- [23] Open Source. Body-Parser library.  
<http://github.com/expressjs/body-parser/>, November 2015.
- [24] Open Source. bootstrap-fileinput library.  
<http://github.com/kartik-v/bootstrap-fileinput/>, November 2015.
- [25] Open Source. busboy-body-parser library.  
<https://github.com/lennym/busboy-body-parser/>, November 2015.
- [26] Open Source. express-session library.  
<http://github.com/expressjs/session/>, November 2015.
- [27] Open Source. Express Skeleton Generator.  
<http://expressjs.com/starter/generator.html/>, November 2015.
- [28] Open Source. fluent-ffmpeg library.  
<http://github.com/fluent-ffmpeg/node-fluent-ffmpeg/>, November 2015.
- [29] Open Source. GridFS-Stream library.  
<http://github.com/aheckmann/gridfs-stream/>, November 2015.
- [30] Open Source. Mongo DB Driver library.  
<http://docs.mongodb.org/ecosystem/drivers/node-js/>, November 2015.

- [31] Open Source. Morgan library. <http://github.com/expressjs/morgan/>, November 2015.
- [32] Open Source. Nodemailer: A utility for sending emails from Node.js applications. <http://github.com/andris9/Nodemailer/>, November 2015.
- [33] MongoDB team. MongoDB Documentation. <http://docs.mongodb.org/manual/>, November 2015.
- [34] MongoDB team. MongoDB Org. <https://www.mongodb.org/>, November 2015.
- [35] Stack Overflow Team and Developers. Stack Overflow Site. <http://stackoverflow.com/>, November 2015.
- [36] WebStorm Team. WebStorm IDE from JetBrains. <http://www.jetbrains.com/webstorm/>, November 2015.
- [37] Marc Wandschneider. *Learning Node.js: A Hands-On Guide to Building Web Applications in JavaScript*. Addison Wesley, 2015.
- [38] Wikipedia. Active Directory. [http://en.wikipedia.org/wiki/Active\\_Directory/](http://en.wikipedia.org/wiki/Active_Directory/), November 2015.
- [39] Wikipedia. AJAX - Asynchronous JavaScript and XML. [http://en.wikipedia.org/wiki/Ajax\\_\(programming\)/](http://en.wikipedia.org/wiki/Ajax_(programming)), November 2015.
- [40] Wikipedia. AngularJS Doc. <http://en.wikipedia.org/wiki/AngularJS/>, November 2015.

- [41] Wikipedia. Apache HBase.  
[http://en.wikipedia.org/wiki/Apache\\_HBase/](http://en.wikipedia.org/wiki/Apache_HBase/), November 2015.
- [42] Wikipedia. Big Data. [http://en.wikipedia.org/wiki/Big\\_data/](http://en.wikipedia.org/wiki/Big_data/),  
November 2015.
- [43] Wikipedia. Database ACID property.  
<https://en.wikipedia.org/wiki/ACID/>, November 2015.
- [44] Wikipedia. Database Index.  
[http://en.wikipedia.org/wiki/Database\\_index/](http://en.wikipedia.org/wiki/Database_index/), November 2015.
- [45] Wikipedia. Hadoop Ecosystem. <http://hadoop.apache.org/>, November  
2015.
- [46] Wikipedia. HAML - A light-weight mark up language.  
<http://en.wikipedia.org/wiki/Haml/>, November 2015.
- [47] Wikipedia. Hypertext Transfer Protocol.  
[http://en.wikipedia.org/wiki/Hypertext\\_Transfer\\_Protocol/](http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol/), November  
2015.
- [48] Wikipedia. IDE - Integrated Development Environment.  
[http://en.wikipedia.org/wiki/Integrated\\_developmen\\_environment/](http://en.wikipedia.org/wiki/Integrated_developmen_environment/),  
November 2015.
- [49] Wikipedia. JavaScript. <http://en.wikipedia.org/wiki/JavaScript>,  
November 2015.

- [50] Wikipedia. jQuery. <http://en.wikipedia.org/wiki/JQuery/>, November 2015.
- [51] Wikipedia. LDAP.  
[http://en.wikipedia.org/wiki/Lightweight\\_Directory\\_Access\\_Protocol/](http://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol/),  
November 2015.
- [52] Wikipedia. Mean Stack.  
[http://en.wikipedia.org/wiki/MEAN\\_\(software\\_bundle\)](http://en.wikipedia.org/wiki/MEAN_(software_bundle)), November 2015.
- [53] Wikipedia. Model-View-Controller.  
<http://en.wikipedia.org/wiki/Model-view-controller>, November 2015.
- [54] Wikipedia. MongoDB: GridFS.  
<http://docs.mongodb.org/manual/core/gridfs/>, November 2015.
- [55] Wikipedia. MongoDB: sharding concept.  
<http://docs.mongodb.org/manual/core/sharding-introduction/>,  
November 2015.
- [56] Wikipedia. MongoDB Wiki. <http://en.wikipedia.org/wiki/MongoDB/>,  
November 2015.
- [57] Wikipedia. NPM Wiki. <https://en.wikipedia.org/wiki/Npm/>,  
November 2015.
- [58] Wikipedia. RDMS.  
[http://en.wikipedia.org/wiki/Relational\\_database\\_management\\_system/](http://en.wikipedia.org/wiki/Relational_database_management_system/),  
November 2015.

- [59] Wikipedia. Secure Socket Layer.  
[http://en.wikipedia.org/wiki/Transport\\_Layer\\_Security/](http://en.wikipedia.org/wiki/Transport_Layer_Security/), November 2015.
- [60] Wikipedia. SPA - Single Page Application.  
[http://en.wikipedia.org/wiki/Single-page\\_application/](http://en.wikipedia.org/wiki/Single-page_application/), November 2015.
- [61] Wikipedia. Structured Query Language.  
<http://en.wikipedia.org/wiki/SQL/>, November 2015.
- [62] Wikipedia. Web Service and classes of web service.  
[http://en.wikipedia.org/wiki/Web\\_service/](http://en.wikipedia.org/wiki/Web_service/), November 2015.

## Vita

Vandana Gambhir was born in Bangkok, Thailand. She received her Bachelor of Arts with Geography (Honors) and Masters in Business Administration from Guru Nanak Dev University in India. She works as a software engineer in Austin, Texas and began graduate studies in Software Engineering at the University of Texas at Austin in August 2012.

Permanent address: [vgambhir@gmail.com](mailto:vgambhir@gmail.com)

This report was typeset with L<sup>A</sup>T<sub>E</sub>X<sup>†</sup> by the author.

---

<sup>†</sup>L<sup>A</sup>T<sub>E</sub>X is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's T<sub>E</sub>X Program.